

FILE COPY

④

ERL-0493-TR

AR-005-959



DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION  
SALISBURY

**ELECTRONICS RESEARCH LABORATORY**

SOUTH AUSTRALIA

**AD-A220 578**

TECHNICAL REPORT

ERL-0493-TR

AN ANALYSIS OF ORDNANCE SOFTWARE USING  
THE MALPAS TOOLS

DR K.J. HAYMAN

APR 11 1990

CH

E

Approved for Public Release

COPY No.

OCTOBER 1989

90 04 10 096

### CONDITIONS OF RELEASE AND DISPOSAL

This document is the property of the Australian Government. The information it contains is released for defence purposes only and must not be disseminated beyond the stated distribution without prior approval.

Delimitation is only with the specific approval of the Releasing Authority as given in the Secondary Distribution statement.

This information may be subject to privately owned rights.

The officer in possession of this document is responsible for its safe custody. When no longer required the document should NOT BE DESTROYED but returned to the Main Library, DSTO, Salisbury, South Australia.

UNCLASSIFIED

AR-005-959

DEPARTMENT OF DEFENCE  
DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION  
ELECTRONICS RESEARCH LABORATORY

TECHNICAL REPORT

ERL-0493-TR

AN ANALYSIS OF ORDNANCE SOFTWARE USING THE MALPAS TOOLS

Dr K.J. Hayman

**Summary**

The increasing use of software in systems where a failure endangers human life is creating an awareness of the need for careful verification of the correct functioning of such software. In this light, an analysis was made of the software to control a "smart" ordnance device, by applying the MALPAS static analysis package. This report presents the results of the analysis, both as they relate to the specific software being analysed and more generally in terms of the experience and insight gained into the application of static analysis techniques to the verification of real-time software. Two significant safety related flaws were detected during the analysis, one in the software itself and the other in the specification of the device. Comparisons are also made between the findings of the analysis and currently proposed standards for writing safety-critical software.

---

POSTAL ADDRESS: Director, Electronics Research Laboratory,  
PO Box 1600, Salisbury, South Australia, 5108.

---

UNCLASSIFIED

## TABLE OF CONTENTS

1. INTRODUCTION	1
2. TRANSLATION OF ASSEMBLY CODE	2
2.1 Modelling microprocessor features	3
2.2 Modelling interrupts	5
2.3 Eight bit arithmetic	6
2.4 "Action" routines	7
3. ANALYSIS OF THE SOFTWARE	7
3.1 The specification	8
3.2 The assembler code	12
3.2.1 Convert.To.BCD	12
3.2.2 Fail	14
3.2.3 Hex.To.BCD	14
3.2.4 Read.Switches	14
3.2.5 ROM	15
3.3 General comments about the code	15
4. SPECIFIC QUESTIONS	16
5. CONCLUSIONS	20
REFERENCES	23

Accession For	
NTIS GCM-1	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Last name/initials/	
Date/Time/Location	
or	
Date/Time/Location	
A-1	

## 1. INTRODUCTION

The increasing use of software in systems where a failure endangers human life is creating an awareness of the need for careful verification of the correct functioning of such software. It is this realisation that has led to the development of such documents as the United Kingdom's draft Defence Standards 00-55 and 00-56(ref.1) and(ref.2) for the development of safety-critical software systems. Works such as(ref.3) also present arguments for and against the use of various languages, language constructs, development techniques, etc, based on the need for the resulting software to be verifiably correct.

This report presents the results of the analysis of a relatively small piece of software which controls a "smart" ordnance device. The analysis was carried out with the aid of the MALPAS verification package(ref.4), which was chosen both because of its speed of analysis and its ready availability.

The study had several major objectives:

- to gain expertise in the use of the MALPAS tools for verifying ordnance software, and hence to evaluate the effectiveness of MALPAS in such a context.
- to gain insight into the restrictions on such things as programming languages specified by documents such as(ref.1). This can then be used to determine whether these restrictions are required or whether they could be relaxed while still allowing verification of the software.
- to analyse the specific software and expose any flaws in programming or design that may not have been uncovered in its development and testing, hence ensuring that no software failure could possibly compromise safety.
- to address two specific questions related to the safety features of the software.

While many of the specifics of(ref.1) may currently not be applicable in an Australian context, the broad thrust of the draft Standard is very relevant. In general it seeks to mandate the use of very careful top-down development of specifications, with each new level being a demonstrably correct instantiation of the preceding one. The choice of programming language is also restricted to those languages (or subsets of languages) with a certified compiler and specific language features (for example, block structured and strongly typed). This is designed to ensure that the compiled code correctly implements the high-level code and also that the final coding is as simple, clear and straight-forward as possible, without any coding "tricks" which perform non-obvious functions. The chosen language (or subset) must also have a formally defined syntax as well as well-defined semantics, so as to disallow the use of any feature whose exact semantic meaning either is not precisely defined or is implementation dependent.

The specific software analysed was that designed to control the EXDET fuse. To quote the "Software Requirement Specification"(ref.5), "EXDET is a completely self contained electronic time delay arming and firing device capable of providing an electrical impulse to an externally connected electric detonator or a percussion impulse to an igniter connected to a non-electric detonator or safety fuse". The code considered here was version 3.0 of the software.

The EXDET software was written without knowledge of (ref. 1) or any similar documents, and as such should not be expected to meet the criteria established there. Many of the concepts, however, such as clear, simple coding without unnecessary obscure "tricks", and clear documentation in the code of all the assumptions on which the correct functioning of the code rests, should be standard practice for all software development. This is especially so for software that is recognised as being safety-critical. Such development and coding not only help to reduce errors in the code, but greatly aid both the verification that the code conforms to its specification, and also any future maintenance of the software which may be required.

The main hardware components of interest in the EXDET device are a Philips PCD3343 microprocessor, which includes 224 bytes of RAM (for data storage) and 3 kilobytes of ROM (for program storage), a clock/calendar chip which is used to provide a second independent time source, and a safety timer, which provides a guaranteed thirty second delay between the arming of the device and detonation.

The remainder of this report looks at the translation of the software into the Intermediate Language used by the MALPAS analysers (Section 2), the results of analysing the code (Section 3), the answers to specific questions posed about the software (Section 4) and the conclusions arising from the analysis (Section 5).

## 2. TRANSLATION OF ASSEMBLY CODE

The first major step in the verification process was to translate the assembly code provided into MALPAS Intermediate Language (IL). In the absence of an automatic translator to perform this task the translation must be done by hand. No benefit was to be gained by writing a translator, there being no perceived use for one beyond the end of the EXDET verification. It was found that the best way to achieve the translation was a two phase process; first implement procedures for each of the microprocessor instructions that are used by the program, and then simply model each instruction of the original assembly code by a call to the corresponding procedure with appropriate parameters.

The PCD3343 microprocessor has one accumulator (referred to as A), a carry flag (C), an eight-element subroutine call stack (located in part of the RAM) and two banks of eight general registers, only one of which is active at a time (R0..R7, also mapped into the RAM). There are several other status flags, which are grouped together with the C flag and the stack pointer into a "processor status word" (PSW), but these are not of interest in this report. Immediate values are specified to the assembler by preceding them with a "#" symbol (for example, #7 denotes the value 7), hexadecimal numbers are written followed by an "H" (so #0CH represents the literal decimal value 12) and indirect addressing is signified by a leading "@" (@R0 refers to the datum whose address is the current contents of register R0).

The major assembler instructions used in the EXDET software are:

Mnemonic		Function
ADD	x,y	Add y to x, putting the result in x
ADDC	x,y	Same as ADD, except add the carry bit as well
CALL	dest	Call subroutine at address dest
CPL	x	Bitwise inversion of x
CLR	x	Clear x (that is, set value x to 0)
DJNZ	x, dest	Decrement x by 1 and jump if result is non-zero
INC	x	Increment x by 1
JBn	dest	Jump to dest if bit n of A is 1
JC	dest	Jump to dest if carry bit is 1
JMP	dest	Unconditional jump to dest
MOV	x,y	Copy data from y to x
RET		Return from subroutine call
RLC	A	Rotate Accumulator left through Carry flag

In this table, "x" and "y" represent data items while "dest" represents a code address within the program. Note that this list covers only those instructions that are explicitly mentioned in this report. It does not cover all the valid instructions of the PCD3343, or even the complete subset used by the EXDET software. Note also that some instructions can change some of the status flags (for example, an ADD instruction can change the carry flag C). Details of such effects are listed in (ref.6).

As well as the above instructions the macro MOVE will be referred to. This is used by the software to generate the special opcodes required for some data transfers (for example, reading data from program memory, transfers to and from I/O ports etc).

In general, no effort was made to optimise the code in any way during the initial translation, so as to preserve, as far as possible, the nature of the original code. The one exception to this was to model the "MOV" instruction by an assignment statement rather than a procedure call. All other simplification was done only after careful examination of the output from the various analysers.

## 2.1 Modelling microprocessor features

The microprocessor's memory can be modelled in one of two ways. Firstly, it could be considered as a single array of 224 integer values, so the IL reference "Memory(I)" would refer to location I in the RAM. This approach has the advantage of being very close to the assembly code's view of the data, and allows easy translation of such features as indirect addressing of memory. The major disadvantage is that the MALPAS information flow analyser is rendered useless (since the result "Memory depends on Memory" is of no practical use). Also the simplifier would be presented with many possibly deeply nested calls to the "update" procedure (which is used to change the value of an array element), especially since the machine registers are memory mapped, meaning that any change in the value in a register would change the memory state. This would slow down and greatly complicate the analysis without any corresponding benefit being gained. Since the information flow analyser is one of the major sources of information about the behaviour of a program, this approach was regarded as unsuitable.

The alternative is to model the memory as distinct variables (or small arrays in cases where this is appropriate). Given this implementation, the MALPAS information flow analyser can provide much valuable data about the information flows within the program, since changes to individual variables can be traced. The disadvantage of this approach is that indirect memory accesses require much more care to correctly translate to IL; in particular to minimise the possibility that errors caused by variable "aliasing" in the assembler code are lost in the translation process. Also, many of the routines, particularly the higher level ones, must now have lengthy parameter lists, since global variables are not currently supported by IL.

In addition to the variables allocated in RAM by the program, such objects as the microprocessor registers, accumulator, I/O ports and status flags had to be considered in the IL model. The initial approach was to model all these items as appropriately typed variables declared in the main program section and passed as parameters through to any other routines that required them.

The translation chosen modelled the status flags as a collection of Boolean variables, so that individual flags could be passed as required. The accumulator was an integer variable. The I/O ports were modelled as integer variables, with specially defined operators to enable tracking of values read/written. The general registers presented a problem, since the PCD3343 has two sets of eight registers, only one of which is active at any one time. To avoid large amounts of complexity, these were initially modelled as three sets of eight integer variables; one for each of the actual sets of registers and one to be the "currently-in-use" set. The machine instruction for switching register sets was modelled by copying the current set to whichever actual bank they were representing and then copying the values for the new set into the current set.

The fundamental disadvantage experienced with this implementation was the large number of parameters that had to be passed to subroutines, particularly through the high-level routines. Because the specification of the routines ended up being derived bottom-up (see Section 3.1 below) this approach led to large and irrelevant information dependencies being carried up to the higher level routines in the form of machine-level variables. This complicated the analysis enormously, to the extent that the higher level routines became impossible to analyse due to the complexity of the resulting expressions.

The solution found for this problem was to model such things as the status flags, registers and I/O ports locally in the routines that use them, wherever this was possible. In doing this, the results from the information flow analyser were important, since such values could only be modelled locally when there was no relevant flow of information through them to any calling routine. In cases where there was such information flow, it is necessary to pass the variable as a parameter from the calling routine, so that the information flow can be accurately modelled. This change in approach simplified the analysis enormously, since large amounts of useless information flow (for example, the "flow" caused by the change in value of a register which a called subroutine used as temporary storage) was removed from the model, allowing analysis to be concentrated on the important and relevant flows. It also allowed instances where there was a non-obvious information flow to or from a routine through such data items to be highlighted.

The obvious danger in this translation strategy lies in code which uses techniques such as variable aliasing or indirect addressing in such a way that assumed information flow

is not obvious. Such programming techniques could easily lead to incorrect modelling of the code, and thus potentially mask errors in the original code. Since examples of this type of programming were found to be present in the EXDET software, extra care had to be taken to avoid such errors in translation. The MALPAS data use and information flow analysers were of use in detecting some errors in this class, since such errors are characterised by uninitialised or badly-used variables. A more detailed discussion of these problems is in Section 3.3 below.

## 2.2 Modelling interrupts

Perhaps the single largest problem with translating the assembler code was modelling interrupts. In the general case, an interrupt can occur at any point in the code after it has been enabled, and the interrupt handler can then proceed to make arbitrary changes to machine registers and/or memory. Such badly behaved interrupt service routines make verification impossible with currently available techniques. As a result, the draft UK standard 00-55(ref.1) restricts the use of interrupts in safety-critical software applications to "a timer interrupt at fixed intervals".

The interrupt handler in the EXDET software, however, was carefully written to avoid such problems. In particular, it has the alternate register bank reserved for its exclusive use, and the value of the accumulator is saved across calls to the handler. The only values written by the routine are never written outside, except to initialise them before interrupts are enabled. All variables read have well-defined values before interrupts are enabled.

While the timing characteristics of the handler are not amenable to being analysed using MALPAS, all paths through the routine appear to be short, so the routine should be able to finish execution well before another interrupt occurs, thus allowing the main-line code time to execute. Any problems of this sort are likely to have been found by dynamic testing, since there are only a small number of semantically possible paths through the handler.

The use of interrupts rather than a polling technique to detect events allows the use of the microprocessor's "IDLE" state. This saves unnecessary power consumption and hence allows the device to operate for much longer periods of time than would otherwise be possible. Advantages such as this, which markedly increase the functionality of the device, make it desirable that techniques be found to verify the safety of code using interrupts, rather than simply banning or severely restricting their use.

Given the design of the EXDET interrupt handler, it becomes possible to apply static code analysis to the routine in similar fashion to any other subprogram. The only additional constraint that must be observed is that an interrupt must not be delivered in the midst of executing a time-critical section of the main code. This can be verified by manually locating any sections of code that are time-critical and ensuring that interrupts are disabled during execution of such code sections.

This experience with interrupts is at odds with the UK approach, which allows only a fixed interval timer interrupt. The results obtained here indicate that it is the interrupt handler's interaction with the mainline code and data, more than the regularity or

otherwise of its occurrence, which determines whether it can be analysed. Also, since only high-level languages may be used under(ref.1), all code must be compiled which makes it very difficult if not impossible to write code which exactly satisfies some time-based design criteria. A more useful approach may be to require the developer to show that any interrupt handler is "well-behaved" and also that the interrupt cannot be triggered in any time-critical section of code. By "well-behaved", it is intended that the routine should have a well defined and, as far as possible, deterministic interaction with the main segments of the code and data. This can be achieved if the handler:

- does not read the value of any variable that may be uninitialised when interrupts are enabled, unless the handler is guaranteed to initialise it first;
- does not change the value of any variable which may also be modified outside the routine while interrupts are enabled; and
- completes execution in less than the expected minimum time between interrupts occurring;

Such an interrupt will have no adverse effect on the running of the main program and can thus be analysed in two separate stages. Firstly, the static properties of the code, such as control flow, data use and the like, can be analysed using a tool such as MALPAS. Then the dynamic properties, such as the timing of interrupts with respect to time-critical sections of code, can be checked.

The verification of interrupts, including the possible revision of the above definition of "well-behaved" handlers to include a more general class of handlers, is a topic for further investigation.

### 2.3 Eight bit arithmetic

Since the PCD3343 is an eight bit microprocessor, all arithmetic operations must be done to eight bit precision, rather than the 32 bit precision of MALPAS integers. The routines written at AWE Aldermaston(ref.7) implemented most of the required functions, and the remainder were easily derived from these routines. Use of these routines, however, resulted in many calls to the simplifier and in fact exceeded the default value of 40 for the DEPTH qualifier of MALPAS. Changing this value to 200 alleviated the problem while still providing a check on any "infinite" recursion of the simplifier.

One problem became apparent with the REPLACE rules for the bit manipulation routines if the first parameter did not have a known explicit value. In this case, the REPLACE rule was invoked, but the resulting (much more complicated) expression could not be simplified, thus adding significant complexity to the final output from MALPAS. This problem is overcome by changing REPLACE rules of the form

```
REPLACE (a,b : integer) a OP b  
BY c;
```

where a, b and c are integer expressions and OP is an binary operator, to the equivalent

```
REPLACE (a,b : integer) a OP b  
BY c IF a >= 0,  
BY c IF a < 0;
```

This has the effect of preventing the application of the REPLACE rule unless an explicit value for *a* is known.

## 2.4 "Action" routines

Another problem with using MALPAS as a verification aid for the EXDET software (or indeed any real-time software) is that some of the routines lack obvious output data items. For example a time delay loop has no effect on the program data, nor does the routine to transmit data out the serial port. These aspects of the code, while clearly being vital to the correct functioning of the software, are execution-time properties which are dependent on the hardware and its environment (albeit influenced by the software). For this reason, it is impossible to completely analyse such code using a tool such as MALPAS, which is a static code analyser that analyses properties of the code and its data independently of its operating environment.

The method chosen to partially overcome this problem was to use an artificial success variable. This variable is an OUT parameter of a routine, and has the final value TRUE if the routine successfully (as far as can be determined statically) performed its intended function, or FALSE if some failure was detected. Thus the success value of the bottom level routines may be based on status values returned from low-level I/O calls, while the success value of higher level procedures can then easily be determined by combining the values returned from all the lower level procedures called. This provides a simple way to allow a static analyser such as MALPAS to analyse routines that do not just manipulate program data, at the expense of having to carefully keep track of the success status.

Time-critical sections of code pose a different problem. One obvious approach would be to maintain a time variable which is incremented appropriately after each machine instruction, so as to accurately reflect the number of machine cycles elapsed. Any time-critical code can then be checked to comply with the desired criteria. Rather than add this significant extra complexity to the model, however, it was decided that any time-critical properties of the code would be verified by hand.

## 3. ANALYSIS OF THE SOFTWARE

Having translated the assembler code into MALPAS IL, the various MALPAS analysers can then be applied to the code. This process tends to be iterative in nature, since output from one run of an analyser may suggest more effective translations of the original assembler code into IL, to enable simpler and more useful output to be generated. The other benefit of making coding optimisations this way, rather than in the initial translation phase, is that by referring to the output from the analysers, it can be shown that the optimisations in no way effect the equivalence of the model and the original code.

### 3.1 The specification

Before anything can be determined about the correctness of the program, there needs to be a well-defined specification for the software. The specification for the EXDET software consists of two documents: the "Software Requirement Specification for the Explosives Detonator"(ref.5) and the "EXDET Software Structured Design: Module Specification"(ref.8). As the titles suggest,(ref.5) is a high-level English language description of the desired behaviour of the software, while(ref.8) is a much lower level pseudo-code specification of the sequence of steps each routine of the software is required to perform. It is notable that there is no attempt to show that the module-level specification(ref.8) satisfies the high-level specification(ref.5), nor were any intermediate-level specifications provided. Thus verifying the software against the module-level specification is only half-way towards verification of the software. It then remains to show that the module specification satisfies the high-level specification.

Detailed examination of the module specification(ref.8) reveals several places where the specification can be shown to be flawed. For example, the following is the specification for the procedure Convert\_Days\_To\_Months, which is used to convert a number of days into a number of whole months and days, based on a starting day of the 1st of January.

```
Raw_Months = 0
IF (Raw_Days - 31 < 0) THEN
  {
    Raw_Days_Left = Raw_Days + 31
    RETURN
  }
Raw_Months = Raw_Months + 1
IF (Raw_Days - 28 < 0) THEN
  {
    Raw_Days_Left = Raw_Days + 28
    RETURN
  }
Raw_Months = Raw_Months + 1
IF (Raw_Days - 31 < 0) THEN
  {
    Raw_Days_Left = Raw_Days + 31
    RETURN
  }
Raw_Months = Raw_Months + 1
IF (Raw_Days - 30 < 0) THEN
  {
    Raw_Days_Left = Raw_Days + 30
    RETURN
  }
```

Leap years can be (and are) correctly and safely ignored, although this assumption is never stated explicitly anywhere in the specification, the program code or its documentation. This is correct behaviour since the months figure is only used to program the

external clock/calendar chip, which expects its delay in months, days, hours, minutes and seconds, where months are based on the first of January in a non-leap year.

Also left implicit is the input constraint of (*Raw\_Days* < 120), presumably since the software (assuming it to be correct) can only be programmed up to 99 days in advance. This omission (and similar ones elsewhere in the specification) may prove disastrous if the software is modified to allow longer delays without a detailed knowledge of such unstated assumptions.

A study of the above specification reveals glaring deficiencies. The value of *Raw\_Days* is never changed. Hence if the first "IF" statement is not satisfied, none of the other IF statements can be satisfied either. This leaves the value of *Raw\_Days\_Left* undefined at the end of the specification. Even if the first IF statement is satisfied, an incorrect value of *Raw\_Days\_Left* is returned.

A look at the corresponding assembler code is revealing.

```

DAYS_TO_MONTHS  CLR    C           ; clear carry bit
                 MOV    RO,#RAW_DAYS ; get address of Raw_Days
                 MOV    A,@RO        ; put value of Raw_Days in A
                 MOV    R1,#0        ; clear R1
                 CPL    A           ; these 3 instructions
                 ADD    A,#31        ; subtract 31 from A
                 CPL    A           ;
                 JC     BACK_STEP_31 ; Jump if Carry set,
                                     ; which means that A was < 31
                                     ; no branch if A was >= 31
                 INC    R1          ; increment number of months
                 CPL    A           ; subtract another 28...
                 ADD    A,#28
                 CPL    A
                 JC     BACK_STEP_28 ; jump if result < 0
                 ...               ; repetitious code removed

OUT_WITH_IT     MOV    RO,#RAW_DAYS_LEFT
                 MOV    @RO,A        ; store A in Raw_Days_Left
                 MOV    RO,#RAW_MONTHS
                 MOV    A,R1
                 MOV    @RO,A        ; store R1 in Raw_Months
                 RET

...

BACK_STEP_31    ADD    A,#31        ; add 31 back to A
                 JMP    OUT_WITH_IT ; clean up and exit
BACK_STEP_28    ADD    A,#28        ; add 30 back to A
                 JMP    OUT_WITH_IT ; clean up and exit

```

It can be seen here that the value A (the accumulator), which is *initially* Raw\_Days, is continuously decreased throughout the routine, until it becomes negative. When this happens, the last subtraction is reversed, the values of Raw\_Days\_Left and Raw\_Months are set and the routine exits. This assembler code can be verified to correctly convert a number of days (up to 119) into whole months and remaining days, based on the obvious, common-sense interpretation of this operation.

The fundamental problem with the specification is that the value of Raw\_Days, on which all the comparisons are done, is never changed, hence the comparisons as written make no sense. Also, the add-back step from the assembly code, which is necessary since the microprocessor has no "compare value" instruction but rather must continually subtract values until the result becomes negative, is also present in the specification. Without the repeated subtraction having been done this leads to an erroneous result.

The effects of this erroneous specification are two-fold. Firstly, sound programming practice for safety critical software development dictates that the module level specification should have been generated from higher-level specifications, and the actual assembler code to implement it designed and coded later. The presence of the add-back operations in the specification suggests that the assembler code may have been written first, with the specification then being reverse-engineered from the assembler code. This in turn suggests that the remainder of the specification may have been similarly derived. If this is indeed the case, then the specification can at best be a higher-level version of the assembler code, complete with any errors that the assembler code may contain. It can thus in no sense be a specification of what the system is *intended* to do, but rather can only be a specification of what the assembler implementation *actually does* do. Such a specification is of no use for the verification of the correct functioning of the software.

Secondly, regardless of whether or not the specification is actually a reverse-engineered version of the assembler, the fact that the specification is so clearly and badly flawed must cast sufficient doubt on the integrity of the specification that it is of no use for verification purposes.

This lack of confidence in the specification means that any flaws detected in the assembler code could just as well be errors in the specification; for instance the first statement in the specification for procedure Run\_Mode is

E\_Test = 0

whilst the assembler code starts with

```
MODE_RUN      MOV     RO,#E_TEST
               MOV     @RO,#01      ; enable test mode
```

These two versions are contradictory, and which is correct is then a matter for a subjective interpretation, based on knowledge of the overall program. This need for such subjective judgement is clearly unsatisfactory when dealing with safety-critical software. In fact, once again, it is the specification which appears to be incorrect rather than the assembler code.

Another aspect of the specification which would have greatly aided the verification process is a concise definition of the exact dependence of the output data on the inputs. Again this is necessary information, since without it the best that can be done is to analyse the code to find the actual dependencies (in effect, reverse engineer the specifications from the code), and then confirm that these are subjectively "reasonable". This is again an undesirable approach when dealing with safety-critical software.

Finally, the detailed functioning of routines such as Serial\_Receive and Serial\_Transmit could not be checked, since no function specification for their operation was provided (other than a passing reference to the "Philips Single Chip Microcomputer Software Examples" in the module specification, a document which was absent from the set of documentation provided for the verification).

Since the module-level specification document(ref.8) is the sole detailed specification for the system, its lack of integrity renders formal verification of the EXDET software impossible. What can be done, however, is to consider the assembler code in bottom-up fashion (starting with routines that call no others) using the MALPAS analysers to derive information-flow and semantic information. This allows us to form a picture of what the software is actually doing. If this information can be simplified sufficiently to allow manual analysis, it can be used together with the English language system specification to check that what the code is doing is subjectively "reasonable". Such analysis is of course extremely informal and prone to missing some errors, especially since the correct behaviour in all circumstances may not be intuitively clear.

Another deficiency in the high level specifications provided concerns the "safety timer". It is stated(ref.5) that detonation will be initiated "on the expiration of the preset delay and the safety timer's thirty second delay", once the device is armed. The specification for the final pre-arming tests also states that there is "... a check to ensure the safety timer is configured correctly ...". It is not, however, stated exactly when this timer is actually started, in particular whether it is at the completion of the final pre-arming test, or when the device is actually armed. Reading the assembler code to determine which has been implemented provides no answers. The safety timer is accessed through the same I/O port as the main rotary switch, and all supporting documents refer only to this use.

Further information was obtained from the manufacturer to fill in the missing details. In fact, the safety timer is controlled almost entirely by hardware, the only output visible to the software being a signal that the delay has expired. The only software control is the initiation of the accelerated test when the unit is powered up. Despite the lack of software control of the safety timer, it would have been extremely useful had some details of its operation been included in the specifications and/or code, since the output signal from this timer forms a critical part of the software's decision on whether or not to detonate.

Overall, this project indicates quite clearly the problems associated with not developing software top-down, starting with a high-level requirements specification and then gradually refining this until the code is written as the last step. In this case it appears that the code was written before at least some of the specification documents, which were later derived at least partially from the code they are intended to define. This invalidates the specification as a reliable source of information of the intended actions

of the software, and thus makes formal verification of the complete software impossible.

### 3.2 The assembler code

The code, having been translated into MALPAS IL as described in Section 1 above, was analysed using the MALPAS analysers, as indicated in Table 1. One problem was caused by the listing itself, which was the printed output from an assembler. The assembler had a limit on the length of output lines which effectively meant that comments longer than about 35 characters were truncated. In some cases, this truncation obscured the meaning of the comments, which made interpreting the intended function of the code harder. Despite this, the comments that were present were generally good and greatly enhanced the readability of the code.

Initially, the control flow analyser was used to determine the structure of the code. This enabled the IL code to be significantly simplified by the appropriate use of IL constructs like IF ... THEN ... ELSE and LOOP ... ENDLOOP. Also a "SUB" procedure (integer subtraction) was implemented to replace the code sequence

```
CPL      A
ADD      A,#n
CPL      A
```

This assembler code can easily be shown to subtract the value "n" from the value in the accumulator A, as all such calculations are effectively unsigned with appropriate checks being made for underflows. Such simplification aids not only other analysers but also greatly increases the readability of the code to humans.

The actual structure of the software is an initialisation phase on powering up the device, followed by a main program loop. This loop examines the state of the rotary switch on the device and calls the appropriate routine to handle the state currently selected. Thus the action of the software after initialisation is determined by examination of the five mode handlers (one for each of "PROGRAM", "STANDBY", "RUN", "TEST" and "ARM" modes) and the handler for the timer interrupt. This partitioning of the software aids analysis by reducing the size of the code to be analysed at once to manageable proportions.

Having determined the structure of the program from control flow analysis, the data use, information flow and semantic analysers can then be used to determine the actual information flows and data dependencies within the code, as well as pointing to any errors made in the translation, as explained earlier. Such analysis leads to the following observations about some of the routines, presented in alphabetical order.

#### 3.2.1 Convert\_To\_BCD

The instruction "CLR C" is redundant. Also, since at least Hex\_To\_BCD expects the tens value returned in R1 to be in the range  $0 \leq R1 \leq 9$ , there should be an input condition of  $0 \leq A < 100$  on input to this routine.

TABLE 1. SUCCESS OF ANALYSERS ON THE VARIOUS ROUTINES

Routine name	Control flow	Data use	Information flow	Semantic analysis	Compliance analysis
CLEAR_DISP	ok	ok	ok	*	-
CONVERT_TO_BCD	ok	ok	ok	ok	x
DAYS_TO_MONTHS	ok	ok	ok	ok	(ok)
DELAY_1.0	ok	ok	ok	*	-
DISPLAY_MSG	ok	ok	ok	*	-
ERROR_LIMIT	ok	ok	ok	ok	-
EXT_INT	null routine				
FAIL	x	ok	ok	*	-
HEX_TO_BCD	ok	(ok)	ok	*	-
INITIALISATION	ok	ok	ok	*	-
MAIN_PROGRAM	ok	ok	ok	*	-
MASK_FILTER	ok	ok	ok	*	-
MODE_ARM	ok	ok	ok	*	-
MODE_PROGRAM	ok	ok	ok	*	-
MODE_RUN	ok	ok	ok	*	-
MODE_STANDBY	ok	ok	ok	*	-
MODE_TEST	ok	ok	ok	*	-
POWER_UP_TEST	ok	ok	ok	*	-
RAM	ok	ok	ok	*	-
READ_SWITCHES	x	ok	ok	*	-
ROM	ok	x	ok	*	-
SERIAL_INT	null routine				
SERIAL_RECEIVE	ok	ok	ok	*	-
SERIAL_TRANSMIT	x	ok	ok	*	-
STOP_IT	ok	ok	ok	*	-
TIMER_CLOCK_INT	ok	ok	ok	ok	-
ZERO_ALARM	ok	ok	ok	*	-
ZERO_TIME	ok	ok	ok	*	-

Note: "x" means that the analyser detected a problem, "\*" indicates that the analyser completed but the results were not fully analysed, "(ok)" means that the routine was shown to be correct after obvious errors had been removed from the specification, and "-" means that the analyser was not run.

### 3.2.2 Fail

This routine calls Serial\_Transmit to display an error code on the liquid crystal display (LCD). This causes no problems, unless Serial\_Transmit detects an error, in which case it calls Fail to display an error code. This potential infinite recursion and its consequences appear not to have been considered, and leads to a possible compromise of safety. Suppose that an error is detected after the device is armed; the correct action is to display an appropriate error code and then call the "STOP" code. If some failure triggers the infinite loop however, the "STOP" code is never reached, and the countdown continues via the timer interrupt service routine. The countdown may then successfully expire, despite the presence of some (detected) error condition which was supposed to stop the processing.

While this problem does not lead to a specific identified compromise of safety, it is considered most undesirable that code which has been identified as being safety-critical should fail to correctly handle a detected error condition. In this case, the correct (and obviously the intended) action is to stop the device, and the code should be modified to ensure that this happens.

### 3.2.3 Hex\_To\_BCD

There is an implicit assumption made by this routine that the registers R6 and R7 contain meaningful values on entry. In general, these are set up by the routine Display\_Msg and are required to remain valid for a sufficient period for the code to work correctly. This is indeed the case in the present software (although this is not obvious to a casual inspection), but this assumption should be clearly stated, so as to avoid the registers being re-used inappropriately in some future modification of the code.

### 3.2.4 Read\_Switches

The assembler code

```

MOV      RO, #F_ARM
MOV      A, @RO
JB7      IN_ARM
...
IN_ARM
```

is used to preform a check which the module level specification describes as

```

IF (F_Arm = FFH) THEN
...
```

Now F\_Arm (like the other F\_\* variables) is initially set by calling Read\_Switches and Mask\_Filter eight times in a loop, each loop filling in a different bit in the variables.

This leaves unanswered the question of whether it is justified to base checks on just one bit of this result, or whether the check should be the more stringent version from the specification despite the obvious additional code required to implement it. Checks on these variables are done in the above "JB7" fashion at many places in the code, and should really either be justified as being sufficient or replaced by a more rigorous alternative.

### 3.2.5 ROM

This routine does not appear to initialise the carry flag before the "RLC A" instruction at line 2370. Reliance on such volatile data seems likely to produce unexpected results. A comment stating (if not justifying) that the initial value of this flag is "don't care" would be desirable, since this fact is far from obvious without detailed consideration of the code.

A high level definition of the CRC used would allow verification of the implementation of the algorithm. Checking the actual expected value would have required both the expected value and a machine-readable version of the code. This latter is arguably unnecessary as having an incorrect expected checksum should lead to a power-up failure very early during the testing of the device.

### 3.3 General comments about the code

In general, the code is clear and straight-forward to analyse, except as noted above. However, the use of code sections such as

```

                                MOV     R3,#04H
                                MOV     A,PSW
                                MOV     RO,#1CH      ; assume register bank 1
                                JB4     NEXTCHAR
                                MOV     RO,#04H      ; set register bank 0
NEXTCHAR                       MOV     A,R1
                                MOVE     A,0A
                                MOV     CRO,A
                                INC      R1
                                INC      RO
                                DJNZ     R3,NEXTCHAR

```

taken from the routine Display\_Msg, rather than the equivalent

```

                                MOV     A,R1
                                MOVE     A,0A      ; read first byte of msg
                                MOV     R4,A      ; store in R4
                                INC      R1
                                MOV     A,R1
                                MOVE     A,0A      ; read second byte

```

```
MOV    R5,A           ; store in R5
INC     R1
MOV     A,R1
MOVE    A,0A           ; read third byte
MOV     R6,A           ; store in R6
INC     R1
MOV     A,R1
MOVE    A,0A           ; read last byte
MOV     R7,A           ; store in R7
```

obscures some of the actual data flow by referencing the registers R4, R5, R6 and R7 via the pointer R0. Such coding is usually written in an attempt to reduce the code size, but it is interesting to note that the second version is in fact smaller in terms of bytes of code than the first (despite the source form consisting of more lines), as well as being much easier to understand and verify. The increased clarity of the second version may also avoid introducing flaws if the software has to be modified, since the routine's use of the registers would be much clearer. No mention of this use of the registers is made in the current version of the software (despite the dependence of the routine Hex.To.BCD on the values placed in these registers by this code).

It is recognised that, unless the data is stored in RAM rather than registers, such code as the first five lines of the current code are necessary to determine the absolute address of the data, which is required by the routine Serial.Transmit. However, the improvement in overall code clarity of the second version above, even with the extra lines to obtain the required absolute address, would greatly outweigh the extra few bytes of total code length that would be generated.

In a similar vein, there are several places in the code where sets of variables are assumed to be allocated in consecutive memory locations. Again, while this has no influence on the correct functioning of the code as it stands, the assumption is never explicitly stated and may lead to problems if the software must be modified by a person who is not extremely familiar with all of the code.

An analysis of the procedure call graph (which shows which procedures are called by which others) shows that there are seven levels of procedures, where a procedure at any given level is only called by procedures at higher levels. It also shows that, apart from the potential infinite recursion discussed earlier, there is no recursion present in the software. It can thus be seen that it is impossible for the microprocessor's eight-level procedure call stack to overflow, although again there is no evidence that this potential problem was ever considered during the design or implementation of the system.

#### 4. SPECIFIC QUESTIONS

Towards the completion of the verification, when it had become obvious that verification of the correctness of the software in its totality was not possible, two more specific questions were posed relating to the safety characteristics of the software.

- *"Could the software cause incorrect operation of the thirty second safety time delay?"*

Details provided about the operation of the thirty-second safety timer show that it is almost exclusively controlled through hardware. In particular, the timer is enabled when the device's selector switch is moved to "TEST" mode and it is reinitialised when the switch is moved to "ARM" mode. Also, from the semantic analysis of the timer interrupt routine, which makes the final decision to detonate, it is clear that all paths which allow detonation require the presence of the "expired" signal from the safety timer. Since everything else is controlled by or preset in the hardware, most importantly the actual length of the delay, this means that the software can in no way cause incorrect operation of the delay.

- *"Could the software cause premature functioning of the device through software-induced error in the set delay time?"*

This is a more complex question, which involves consideration of the results from data use, information flow and semantic analysis of several sections of the software. The delay is set up in "PROGRAM" mode, but no timers are started until "RUN" mode, and the final delay time is not passed to the independent clock/calendar until the device is armed. Thus to answer the question, it is necessary to determine whether the delay is correctly read in and stored, and also whether this delay is correctly made available to and used by later routines to initialise the timers.

Using the results from the MALPAS analysers in combination with the original code shows that the preset delay is read in and stored in three variables, "Raw\_Days", "Raw\_Hours" and "Raw\_Minutes". These values are initially set to zero days, zero hours and one minute, and can be changed by firstly pressing the "SELECT" button to choose which quantity to change and then the "INCREMENT" button repeatedly to set the desired value. Each time one of these variables is incremented its new value is displayed on the LCD, thus ensuring that the operator is correctly informed of the currently programmed delay setting. In the event of a failure of the LCD display, the error generated by trying to display a value will be detected by the routine Serial\_Transmit, which should ideally cause the device to shut down. As explained above, however, the current effect is for the software to go into an infinite loop. Since the only way out of this loop is to power down the device, this is a safe effect, so failure of the LCD can be disregarded from a safety point of view. Thus when the device is switched from "PROGRAM" mode to "STANDBY" mode, the last value displayed on the LCD matches the internally stored delay.

"STANDBY" mode will read the Raw\_xxx variables if either of the push buttons is activated. In this case, the preset delay is displayed on the LCD, for further manual verification if this is desired. These values are copied into general registers to be passed to the display routines, but data use analysis shows that they are never written to, so there can be no compromise of safety while the device is in this mode.

"RUN" mode is used to initialise the required timers, although the device is not armed at this stage. Data use analysis shows that the Raw\_xxx variables are again only read in this mode, so there is no corruption of the preset delay figures. The Raw\_xxx values are copied directly into another set of variables, named

Tim\_Day, Tim\_Hour and Tim\_Minute, and two variables, Timer\_Min\_Msb and Timer\_Min\_Lsb, are initialised to represent fractions of minutes. All these variables are used by the software to implement the countdown timer that forms one of the time sources for the device. Until the countdown expires (either with the device armed, which may result in detonation, or with it unarmed which merely displays a message on the LCD) or an error is detected, a timer interrupt occurs sixteen times per second.

Semantic analysis of the interrupt handler shows that the code to implement the countdown is correct (given that the interrupts are generated at the correct frequency, which is controlled by hardware). Thus the countdown timer will correctly expire after the preset delay time, starting from when interrupts are enabled. This is done at the end of the code to handle "RUN" mode. Also verified by semantic analysis of this routine is the fact that all three of the software timer, the independent hardware timer and the safety timer must correctly expire before detonation can occur.

Also in the "RUN" mode (but before timer interrupts are enabled), the independent clock/calendar is initialised. This device is reset to zero, and this resetting is checked by a read-back step, with the value being verified as zero. Failure causes the device to shut down. The alarm point of the device is similarly cleared and verified. The count-up of this device is enabled at the same time as the software countdown timer is enabled, but the alarm time is not set until the device is armed. This is another safeguard against premature detonation, since until the alarm time is set, the clock/calendar timer cannot successfully complete, and thus detonation cannot occur.

The routine to handle "TEST" mode does not access either the preset delay (Raw\_xxx) variables or the current countdown (Tim\_xxx) values in any way, according to the data use analysis. Thus the code is of no interest to answering the current question and is not analysed further.

The final point at which the software could conceivably cause premature detonation is in "ARM" mode, when the alarm time is calculated and sent to the independent clock/calendar. Data-use analysis shows that this routine reads the Raw\_xxx variables to obtain the preset delay time. Data use analysis has shown that these have not been altered since they were set in "PROGRAM" mode (except for one case which will be discussed in detail shortly), so these values still accurately reflect the programmed and displayed delay time. The Raw\_Days value is split into Raw\_Months and Raw\_Days\_Left, using the Convert\_Days\_To\_Months routine discussed earlier. As stated there, the assembly language version of this routine is correct for the currently allowed range of allowed delays (up to 99 days). Thus this conversion will complete successfully. These delay values are then converted to BCD using the Convert\_To\_BCD routine, which semantic analysis has shown to be correct for the range of possible values of these variables. These BCD values are then used to set the alarm time for the independent clock/calendar, and the correct setting is verified by reading back the values and comparing them with the originals, failure causing the device to shut down.

The remaining consideration is what happens if the modes are not entered in exactly the expected order. An examination of data use analysis shows that if

"PROGRAM" mode is entered from "STANDBY" mode, or "STANDBY" is entered from "RUN", then the programmed delay time is reset to the default time of one minute. This is the only time, apart from the delay entry sequence in "PROGRAM" mode, that the *Raw\_xxx* variables are modified by the code, as shown by data use analysis. Despite the "reset to one minute" being part of the Requirement Specification(ref.5), it is contended that this behaviour is unsafe, since the LCD remains clear, with the new delay never being displayed unless one of the push buttons is activated in either "STANDBY" or "RUN" mode. Thus in this situation, the actual programmed delay no longer matches the last displayed delay, and the device may well detonate before it is expected, which is a clear safety hazard. Given that(ref.5) specifies this behaviour in order to conform with the Operator Interface requirements of the Hardware Design Specification(ref.9), this hazard must initially be addressed via a review of(ref.9), with changes flowing through to(ref.5) and ultimately to the code.

This problem may be overcome in a number of ways. For example, the idea of a "default delay" could be abandoned, with the operator being forced to enter a delay at any stage that the default delay is currently set. Alternatively, the default delay could be made to be the longest possible delay for the device (currently 99 days, 23 hours and 59 minutes) rather than the shortest. This would mean that even if the delay was reset, the resulting delay would be at least as long as that programmed, so detonation would never be premature. Another possibility is that part of the sequence of initiating the timers in "RUN" mode could be to display the programmed delay. The operator could even be required to activate one of the push buttons after the delay is displayed, as an indication of acceptance.

As a side-effect from the analysis, the actual time of detonation of the device can be determined. The software timer and clock/calendar devices are both initialised when the device is switched into "RUN" mode, so this is a convenient base time to consider as zero. The preset delay time,  $Dt$ , which matches the last time displayed on the LCD (except for reset case described above) is the expected time of detonation. However, the clock/calendar is not programmed with its final time until "ARM" mode is entered at time  $At$ , and at this time the safety timer begins its timing as well (nominally thirty seconds, but in practice it may be slightly longer,  $St$ , where  $30 \leq St \leq 55$  seconds according to the hardware specification). Thus detonation actually occurs at the later of  $Dt$  and  $(At + St)$ . Since  $Dt \geq At$  (or else the software timer completes with the device unarmed and detonation is inhibited), it can be seen that, even given completely accurate timing, detonation may be delayed for up to  $St$  seconds beyond the stored delay time. This behaviour is in accordance with the Requirement Specification(ref.5), and is also intuitively "safe" (that is, detonation is delayed by at least the duration of the safety timer once the device is switched into "ARM" mode.)

The answer of the original question can now be given. The analysis has revealed that as long as the device is stepped through each mode in the "correct" order, the code correctly stores the preset delay and passes this value to the two independent timers. Also, the requirement that each of these timers must expire correctly (as well as the safety-timer) before detonation is allowed is correctly enforced by the code. If, however, the device is returned to "PROGRAM" or "STANDBY" mode having been taken to another mode, the preset delay may be reset to one minute

without this being displayed on the LCD. Since this is the smallest possible delay for the device, it is likely to be shorter than the programmed delay, so when the device is finally armed detonation may occur well before it is expected.

## 5. CONCLUSIONS

Overall, the EXDET software appears to have been well written and tested in terms of conventional programming practices, as shown by the small number of problems detected in the code. While good programming practices certainly helped the verification, it is clear that the code was not written to be verified. Such coding practices as using an indirect reference in a loop to load several machine registers and the many undocumented assumptions (such as certain variables being located in the correct order in consecutive memory locations) on which the correct functioning of the code rely should not have been present if one of the main goals was the verification of the completed code. In conventional software development, however, where the code is checked to be correct by dynamic testing, such coding is (arguably) less of a problem, since the main concern is the correctness of the final answer, with the fine details of the calculation of that answer being, in general, only of secondary importance.

There were two significant problems detected, one being in the software itself while the other was in the specification. It is worth noting, however, that even the specification error was detected from the MALPAS analysis of the code, which indicated behaviour which seemed intuitively unsafe in some cases. The software error was the potential infinite loop of the Fail and Serial-Transmit routines, and the required situation for this to happen is very unlikely to occur (which is why this was not found by dynamic testing of the software). Nevertheless, in a recognised safety-critical context, any detected error must cause the device to enter a safe state until the error can be corrected, since ignored or incorrectly handled error conditions lead to unacceptable risks of a compromise of safety. This problem thus needs to be addressed to ensure that there is no such compromise of safety possible in the EXDET software. The specification error was that the programmed delay could be reset to one minute without the operator necessarily being aware of the change. This also must be corrected as a high priority, since it leads to detonation before the expected time.

In contrast to the generally good quality of the code, however, the specification of the system as presented is entirely inadequate and cannot form the basis for formal verification of the software. It has been shown that

- the module-level specification contains fundamental and obvious errors in simple routines;
- it may well have been reverse-engineered from the code; and
- no attempt was made to show that it conformed to the top-level software specification.

Since a verification can only be as good as the specification against which the software is compared, any one of these points would make verification impossible.

In order to be able to verify the software as being correct, it would be necessary, as well as fixing the errors detected in the software, to develop a series of credible specifications. This should start with a (possibly revised) version of the requirements specification(ref.5), which would clearly, completely and unambiguously specify the desired behaviour of the software. It should end with a module-level specification which would specify for each routine (as a minimum):

- a clear, concise and complete description of its intended function,
- a complete list of any input and/or output parameters,
- a complete list of any global objects that are referenced, including the type of access (such as read and/or written),
- a complete functional specification of any outputs (parameters or globals) in terms of the inputs, and
- a complete list of any assumptions on which the correct functioning of the code is based.

Given such a series of specifications, there would then be a high degree of confidence that the module-level specification accurately represented the top-level specification. This immediately leads to greater confidence that software that meets the specifications is going to meet the user's requirements. Even if such a series of specifications were to be developed for the EXDET software, it would almost certainly be necessary to modify substantial portions of the code so that its compliance with the specification could be assured (even after fixing the problems identified by this study). This would clearly require a large amount of effort, and is a good illustration of the need to consider safety at all stages of the design and development of safety-critical software.

In a more general light, the MALPAS tools have been found to be practical to apply to a significant sized piece of software which controls a real-time device (of the order of 2500 lines of assembler code). In the course of the analysis, useful techniques and "tricks" for applying MALPAS to such software have been demonstrated.

It is apparent from the results of this study that some of the restrictions imposed by the UK Standard 00-55(ref.1) are much stronger than they need to be. It is possible, with good programming discipline, to write good, verifiable code in assembly language using interrupts, just as it is possible to write obscure, even unverifiable, code in "approved" languages. Also, serious consideration must be given to whether the current level of validation of compilers is sufficient for the code they produce to be trustworthy enough for use in safety critical applications. Given good code development and verification practices for assembler code, it may well be that such code is in fact at least as trustworthy, if not more so, than code compiled from a higher level language. While some restrictions are obviously required (for instance, parts of a language whose semantics are machine or compiler dependent should clearly not be allowed), banning techniques, languages and the like is an over-reaction. Rather, we should be seeking to find the broadest possible guidelines that allow for the verifiably safe use of such techniques.

One area where particular attention needs to be paid during the design and development of the software is that of writing specifications. An inadequate specification forces

analysis of the code in a bottom-up fashion, which has two main disadvantages. Firstly, the best result that can be obtained is a statement of what the code actually does, which may be different from what it was intended to do. Secondly, particularly in cases such as assembly code where many objects are globally accessible, large amounts of extraneous detail tends to be carried up through the analysis. This leads to complication of the upper levels of the generated specification, which may completely overwhelm the desired results. If the analyst prunes some information out during the analysis to prevent this happening, there is a possibility that some relevant data will be removed as well, which renders the analysis invalid. Thus a credible and complete specification is essential for the meaningful verification of any software.

The final observation made from the analysis is that, although the specification was inadequate and hence the software could not be verified as being correct, use of the MALPAS tools did help to expose significant problems in both the code and the device's specifications. The potential infinite loop was a subtle problem with only a small chance of occurring even under field conditions. In a controlled test situation over a relatively short time the chances of it occurring were even more remote, and indeed the problem was obviously not found during dynamic testing of the device. This clearly illustrates the worth of using static code analysis to augment conventional dynamic testing before the acceptance of safety-critical software. The specification problem shows the need for careful examination of the output from packages such as MALPAS. Even though the code is "correct" according to the software specification, and this is in turn a "correct" refinement of the hardware design specification, human analysis of the MALPAS output led to the discovery of a weakness in the underlying hardware design specification.

## REFERENCES

No.	Author	Title
1	-	"Requirements for the Procurement of Safety Critical Software in Defence Equipment". Draft Interim Defence Standard 00-55. UK Ministry of Defence, May 1989
2	-	"Requirements for the Analysis of Safety Critical Hazards". Draft Interim Defence Standard 00-56. UK Ministry of Defence, May 1989
3	Cullyer, W.J. and Goodenough, S.J.	"The Choice of Computer Languages for use in Safety-Critical Systems". RSRE Memorandum No 3946, October 1987
4	-	MALPAS Verification Suite, Static Analysis Tools, RTP Software Ltd., Farnham, UK
5	Fairey Australasia Pty Ltd	"Software Requirement Specification for the Explosives Detonator". Document No. FAL/ED/122 Issue 1, October 1988
6	-	"CMOS microcontroller for Telephone Sets". Philips PCD3343 Development Sample Sheet, pp 165-204, August 1984
7	-	Replacement Rules (AWE Aldermaston), Attachment 2 to RTP/5091/1/2, Minutes of MALPAS User Group Meeting, 19th January 1988
8	Fairey Australasia Pty Ltd	"EXDET Software Structured Design: Module Specifications". October 1988
9	Fairey Australasia Pty Ltd	"Requirements Specification FAL/ED/0044, Issue 2".

# DISTRIBUTION

No of Copies

## DEPARTMENT OF DEFENCE

### Defence Science and Technology Organisation

Chief Defence Scientist

First Assistant Secretary Science (Policy)

First Assistant Science Corporate Manangement

Counsellor, Defence Science, London

Counsellor, Defence Science, Washington

Director, DSD

### Electronics Research Laboratory

Director, Electronics Research Laboratory

Chief, Electronic Warfare Division

Chief, Communications Division

Chief, Information Technology Division

Research Leader, Information Technology

Head, Trusted Computer Systems Group

Head, Software Engineering Group

Head, Information Systems Research Group

Head, Command Support Systems Group

Principal Research Scientist, Architectures Team

Principal Engineer, VLSI Team

}

1

Cnt Sht Only

Cnt Sht Only

1

1

1

1

1

1

1

2

1

1

1

1

ERL-0493-TR

Dr M. Anderson, Trusted Computer Systems Group	1
Mr J. Christie, Trusted Computer Systems Group	1
Mr M. Stevens, Trusted Computer Systems Group	1
Mr J. Grundy, Trusted Computer Systems Group	1
Ms K. Eastaughffe, Trusted Computer Systems Group	1
Dr A. Cant, Trusted Computer Systems Group	1
Aeronautical Research Laboratory	
Director, Aeronautical Research Laboratory	1
Surveillance Research Laboratory	
Director, Surveillance Research Laboratory	1
Materials Research Laboratory	
Director, Materials Research Laboratory	1
Weapons Systems Research Laboratory	
Director, Weapons Systems Research Laboratory	1
Navy Office	
Navy Scientific Adviser	Cnt Sht Only
Army Office	
Scientific Adviser - Army	1
Air Office	
Air Force Scientific Adviser	1
Joint Intelligence Organisation (DSTI)	1
Australian Ordnance Council	1

## Libraries and Information Services

Librarian, Technical Reports Centre, Defence Central Library, Campbell Park	1
Document Exchange Centre	
Defence Information Services Branch for:	
Microfiche copying	1
United Kingdom, Defence Research Information Center	2
United States, Defense Technical Information Center	12
Canada, Director, Scientific Information Services	1
New Zealand, Ministry of Defence	1
National Library of Australia	1
Main Library, Defence Science and Technology Organisation Salisbury	2
Library, Aeronautical Research Laboratory	1
Library, Materials Research Laboratory	1
Librarian, DSD, Melbourne	1
Author	2
Spares	6
Total number of copies	60

# DOCUMENT CONTROL DATA SHEET

Security classification of this page :

UNCLASSIFIED

<b>1 DOCUMENT NUMBERS</b>  AR Number : AR-005-959  Series Number : ERL-0493-TR  Other Numbers :	<b>2 SECURITY CLASSIFICATION</b> a. Complete Document : Unclassified b. Title in Isolation : Unclassified c. Summary in Isolation : Unclassified  <b>3 DOWNGRADING / DELIMITING INSTRUCTIONS</b> Limitation to be reviewed in October 1992				
<b>4 TITLE</b>  AN ANALYSIS OF ORDNANCE SOFTWARE USING THE MALPAS TOOLS					
<b>5 PERSONAL AUTHOR (S)</b>  Dr K.J. Hayman	<b>6 DOCUMENT DATE</b> October 1989  <b>7</b> <table border="1"> <tr> <td>7.1 TOTAL NUMBER OF PAGES</td> <td>23</td> </tr> <tr> <td>7.2 NUMBER OF REFERENCES</td> <td>9</td> </tr> </table>	7.1 TOTAL NUMBER OF PAGES	23	7.2 NUMBER OF REFERENCES	9
7.1 TOTAL NUMBER OF PAGES	23				
7.2 NUMBER OF REFERENCES	9				
<b>8</b> <table border="1"> <tr> <td>8.1 CORPORATE AUTHOR (S)</td> <td>Electronics Research Laboratory</td> </tr> <tr> <td>8.2 DOCUMENT SERIES and NUMBER</td> <td>Technical Report 0493</td> </tr> </table>	8.1 CORPORATE AUTHOR (S)	Electronics Research Laboratory	8.2 DOCUMENT SERIES and NUMBER	Technical Report 0493	<b>9 REFERENCE NUMBERS</b> a. Task : b. Sponsoring Agency :  <b>10 COST CODE</b> 255
8.1 CORPORATE AUTHOR (S)	Electronics Research Laboratory				
8.2 DOCUMENT SERIES and NUMBER	Technical Report 0493				
<b>11 IMPRINT (Publishing organisation)</b>  Defence Science and Technology Organisation Salisbury	<b>12 COMPUTER PROGRAM (S)</b> (Title (s) and language (s))				
<b>13 RELEASE LIMITATIONS (of the document)</b>  Approved for Public Release					

Security classification of this page :

UNCLASSIFIED

Security classification of this page :

UNCLASSIFIED

14 ANNOUNCEMENT LIMITATIONS (of the information on these pages)

No limitation

15 DESCRIPTORS

a. EJC Thesaurus  
Terms

- MALPAS (computer program) ;  
Computer program verification ;  
Electric fuzes (ordnance) ;  
Computer program reliability ;  
Safety and arming (ordnance) ;

b. Non - Thesaurus  
Terms

*some delay fuzes*

- EXDET fuze

16 COSATI CODES

0062B

17 SUMMARY OR ABSTRACT

(if this is security classified, the announcement of this report will be similarly classified)

The increasing use of software in systems where a failure endangers human life is creating an awareness of the need for careful verification of the correct functioning of such software. In this light, an analysis was made of the software to control a "smart" ordnance device, by applying the MALPAS static analysis package. This report presents the results of the analysis, both as they relate to the specific software being analysed and more generally in terms of the experience and insight gained into the application of static analysis techniques to the verification of real-time software. Two significant safety related flaws were detected during the analysis, one in the software itself and the other in the specification of the device. Comparisons are also made between the findings of the analysis and currently proposed standards for writing safety-critical software.

Security classification of this page :

UNCLASSIFIED

The official documents produced by the Laboratories of the Defence Science and Technology Organisation Salisbury are issued in one of five categories: Reports, Technical Reports, Technical Memoranda, Manuals and Specifications. The purpose of the latter two categories is self-evident, with the other three categories being used for the following purposes:

- Reports : documents prepared for managerial purposes.
- Technical Reports : records of scientific and technical work of a permanent value intended for other scientists and technologists working in the field.
- Technical Memoranda : intended primarily for disseminating information within the DSTO. They are usually tentative in nature and reflect the personal views of the author.